

Visual Perception Engine: Fast and Flexible Multi-Head Inference for Robotic Vision Tasks

Jakub Łucki^{1,2}, Jonathan Becktor¹, Georgios Georgakis¹, Robert Royce¹ and Shehryar Khattak¹

Abstract—Deploying multiple machine learning models on resource-constrained robotic platforms for different perception tasks often results in redundant computations, large memory footprints, and complex integration challenges. In response, this work presents Visual Perception Engine (VPEngine), a modular framework designed to enable efficient GPU usage for visual multitasking while maintaining extensibility and developer accessibility. Our framework architecture leverages a shared foundation model backbone that extracts image representations, which are efficiently shared, without any unnecessary GPU-CPU memory transfers, across multiple specialized task-specific model heads running in parallel. This design eliminates the computational redundancy inherent in feature extraction component when deploying traditional sequential models while enabling dynamic task prioritization based on application demands. We demonstrate our framework’s capabilities through an example implementation using DINOv2 as the foundation model with multiple task (depth, object detection and semantic segmentation) heads, achieving up to 3x speedup compared to sequential execution. Building on CUDA Multi-Process Service (MPS), VPEngine offers efficient GPU utilization and maintains a constant memory footprint while allowing per-task inference frequencies to be adjusted dynamically during runtime. The framework is written in Python and is open source with ROS2 C++ (Humble) bindings for ease of use by the robotics community across diverse robotic platforms. Our example implementation demonstrates end-to-end real-time performance at ≥ 50 Hz on NVIDIA Jetson Orin AGX for TensorRT optimized models.

I. INTRODUCTION

VISUAL perception is a cornerstone of autonomous robotic system applications, enabling mobile machines to interpret complex environments [1], make informed decisions under uncertainty [2], and interact seamlessly within dynamic real-world scenarios [3]. The ability to extract meaningful information from visual data directly impacts a robot’s ability for navigation, manipulation, obstacle avoidance, and human-robot interaction, making robust perception systems essential for reliable autonomous operations.

Recent advances in computer vision have witnessed a paradigm shift toward visual foundation models that can serve multiple perception tasks through shared representations. Vision transformers trained on internet scale data like DINOv2 [4], SAM [5], and CLIP [6] have demonstrated remarkable capabilities in replacing specialized solutions across diverse

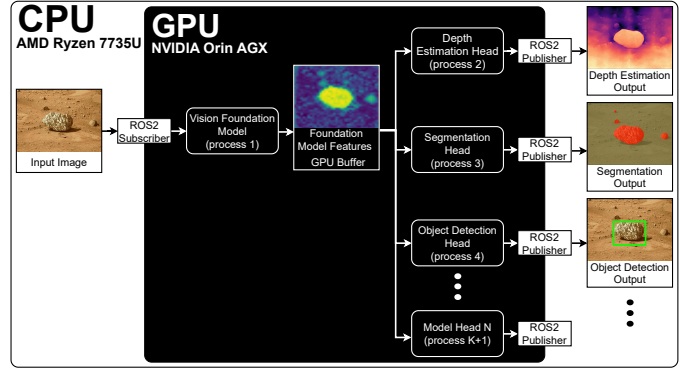


Fig. 1. High level architectural overview of the VPEngine framework.

applications. For instance, DINOv2 extracts image representations that generalize across domain and as such can be used for a variety of vision tasks without any finetuning.

This convergence toward unified architectures suggests that leveraging one large network for multiple perception tasks should be the natural progression for robotic systems. However, current robotic perception frameworks remain inadequate or inefficient in their implementation. For example, NVIDIA Isaac ROS [7], while providing excellent optimized perception capabilities, deploys separate specialized models for each task rather than utilizing a shared foundation model backbone. Isaac ROS, among others, offers individual nodes for stereo depth estimation (ESS), object detection (YOLOv8), and semantic segmentation (Segformer), each running independently. This approach, while functionally robust, perpetuates computational redundancy, as unified vision features can be used for multipurpose downstream tasks [4].

To address these limitations, we introduce Visual Perception Engine (VPEngine), a framework specifically designed to harness the power of visual foundation models for robotic multitasking efficiently. VPEngine uses highly optimized TensorRT models¹, running in separate processes, which combined with CUDA MPS results in a very fast inference and high GPU utilization. Furthermore, we devised custom Inter-Process Communication structures using CUDA API to enable *by reference* GPU memory transfers across processes, which are not supported on Jetson devices. As a consequence, the latency of sharing image representations between the backbone and model heads is negligible. The framework is written in Python, heavily leverages abstract classes and is highly modularized which enables fast prototyping and fosters extendability. A

¹ Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, USA.

² Swiss Federal Institute of Technology (ETH Zürich), Zürich, Switzerland. The research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration (80NM0018D0004).

©2025. All rights reserved.

¹However, it supports native PyTorch as well, since it is not always feasible to convert a model to TensorRT

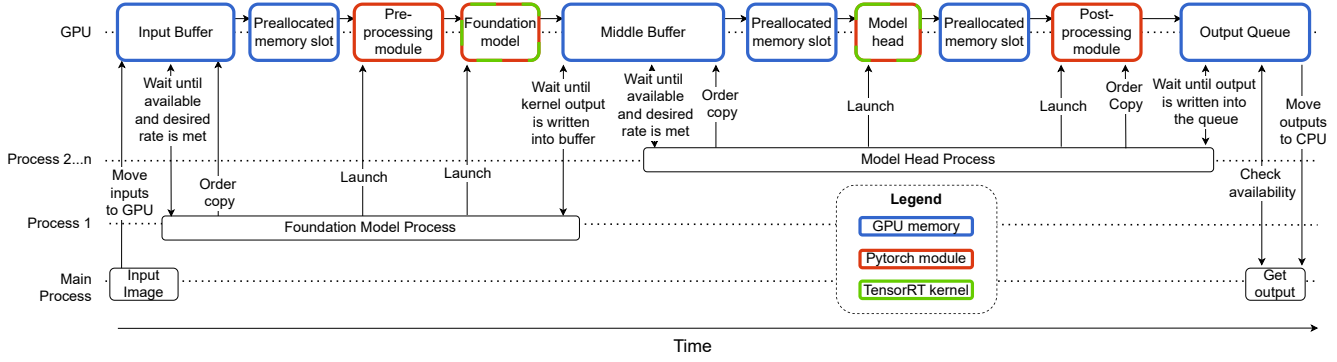


Fig. 2. Alternative High level architectural overview of the VPEngine framework.

high-level overview of the system is shown in Figure 1.

II. DESIGN REQUIREMENTS

To address the challenges of efficient visual multitasking in resource-constrained robotic systems, we establish the following design requirements that guide VPEngine’s architecture and implementation:

- **R1: Fast Inference** - Latency of the perception loop can be the upper bound on the robots reaction speed to a change in the environment. Furthermore, if desired model frequencies are fixed, inference efficiency allows for more GPU time available for other tasks.
- **R2: Memory Predictability** - Autonomous systems must operate reliably for extended periods without memory-related failures that could compromise mission success.
- **R3: Extendability and Flexibility** - Robotic applications vary in their perception needs, requiring flexible architectures that accommodate diverse task combinations.
- **R4: Dynamic Task Prioritization** - Environmental conditions during robot missions, requiring adaptive perception systems that can prioritize critical tasks based on current context.

III. SYSTEM ARCHITECTURE

VPEngine consists of a foundation model (FM) serving as the computational core that provides rich visual representations, which are then efficiently shared and processed by an arbitrary number of lightweight task-specific model heads. Compared to independent task-specific models, this architecture eliminates redundant feature extraction in the earlier stage of each network by sharing of FM visual features among tasks in a memory efficient way without creating feature copies. Detailed system overview is presented in Figure 2.

A. Modular Architecture

VPEngine implements two distinct module types to achieve efficient visual multitasking:

1) *Foundation Module*: The foundation module serves as the shared backbone for all perception tasks, addressing **R1** by centralizing the extraction of features. It consists of four main components: a) an input buffer that receives raw sensor data, b) pre-processing transformations that normalize inputs for the foundation model, c) the foundation model to extract visual features, and d) a middle buffer that stores computed features for sharing among task heads.

The foundation module operates continuously, processing incoming images and maintaining a constant buffer of feature representations. All computations remain strictly on GPU to minimize memory transfer overhead.

2) *Head Module*: Task-specific head modules take FM features as input and perform specialized perception tasks, directly supporting **R3**. Each head module contains a) an interface to asynchronously accesses the middle feature buffer, b) the lightweight task-specific model, c) postprocessing transformations to format outputs for downstream consumers, and d) an output buffer that delivers results to external systems e.g. the ROS2 publisher. Each head modules operate independently, allowing each task head to execute at variable frequency based on the application requirements.

3) *Transforms*: Pre-processing and postprocessing transforms serve as adapters between user data and the core of the engine, accommodating varying tensor shapes, and data types. These lightweight operations ensure compatibility between foundation models and task heads without requiring modifications to the core neural networks.

Transform operations are implemented as simple tensor manipulations (reshaping, normalization, type conversion) that execute efficiently on GPU without significant computational overhead. This abstraction layer enables seamless integration of models with different input/output specifications.

4) *Module separation*: Each model component executes in a separate process to maximize GPU utilization through CUDA MPS, addressing both **R1** and **R3**. To utilize CUDA MPS, a standard MPS server is configured on the deployment device, after which all CUDA-enabled processes automatically leverage shared GPU resources. Additional benefit of using CUDA MPS is that it allocates only one CUDA context for all processes [8].

This multi-process design enables true parallelization when individual models cannot fully saturate GPU resources, a

common scenario when portions of models execute in PyTorch rather than optimized TensorRT kernels. The framework also provides fault isolation where failure in one task head does not compromise the entire perception pipeline.

While multi-process deployment introduces PyTorch initialization overhead across processes, the benefits of better encapsulation, fault isolation, and parallel execution through CUDA MPS significantly outweigh these costs in practice.

5) *Decentralization*: Each module operates independently with minimal central coordination, supporting **R4**. Modules execute as soon as input data becomes available, subject only to their configured frequency limits. Runtime parameters such as execution frequency can be adjusted through the main process coordination system, but modules maintain autonomous operation for maximum responsiveness.

This decentralized approach ensures that high-priority tasks are not blocked by slower components and enables adaptive resource allocation based on changing environmental demands.

B. Memory Management

VPEngine implements ‘by reference’ GPU memory sharing, directly addressing **R2** and contributing to **R1**. This is not supported on Jetson platforms by default hence we created our own Inter-Process Communication structures leveraging low-level CUDA API, in particular function `cuMemImportFromShareableHandle`. As a consequence, foundation model outputs remain GPU-resident, with task heads accessing shared memory regions directly via CUDA pointers. This design eliminates costly GPU-CPU-GPU memory transfers that create overhead for traditional multi-model pipelines.

Memory copies only occur when removing elements from queues/buffers into the pre-allocated processing slots. This copy operation is necessary to release queue slots for new data; otherwise, slots would remain locked until model heads complete processing, potentially causing pipeline stalls.

Memory allocation follows a static pattern established during initialization, with all queue and buffer sizes specified in the configuration file (*config.json*). Each queue and buffer pre-allocates required GPU memory during system startup, ensuring constant memory footprint during operation and preventing runtime allocation failures that could compromise system stability (**R2**).

C. Data Flow

Inter-process communication utilizes two data structures:

- 1) *GPU Queues*: FIFO structures for ordered data transfer between FM and task heads. Queues signal overflow conditions when consumers cannot keep pace with producers.
- 2) *GPU Buffers*: Circular buffers that overwrite oldest data when full, ensuring that the models always access the most recent tensors. This design prioritizes temporal relevance over processing completeness.

All tensors are organized as labeled dictionaries, enabling flexible data routing. Task heads can selectively consume relevant FM features from the complete output dictionary without index-based coupling.

D. Dynamic Frequency Control

Each task head supports runtime frequency adjustment through a control interface. This enables adaptive perception where task execution rates adjust based on environmental changes, computational load, or mission requirements (**R4**). For example, obstacle detection might increase frequency in dense environments while reducing semantic segmentation rate to balance computational resources for robotic tasks.

E. Model Optimization

The system leverages NVIDIA TensorRT for inference optimization, directly supporting **R1** and **R2**. TensorRT compiles neural networks into optimized execution engines that maximize GPU throughput through kernel fusion, precision optimization, and memory layout optimization. This compilation process is particularly crucial for foundation models where inference latency directly impacts the entire pipeline’s performance. Please note in our framework allows for FM and model heads to be independently TensorRT compiled while maintaining efficient FM feature sharing between them

F. Model Management

A unified model registry manages all foundation models and task heads, including version control, metadata tracking, and deployment configuration. The registry supports automatic TensorRT compilation, model validation, and deployment verification across different hardware targets. VPEngine supports both TensorRT-optimized engines for maximum efficiency and native PyTorch models for development flexibility.

IV. EXAMPLE IMPLEMENTATION

In our example implementation, we use three model heads: DepthAnythingV2 (DAV2) [9] responsible for monocular depth estimation (we used optimized code from [10]) compiled with TensorRT, linear layer for semantic segmentation [4] compiled with TensorRT, and a custom FasterRCNN head for 2-class object detection. For foundation model we use ViT-S version of DINOv2 [4] to get rich visual representations. We configured it to output final features along with representations computed at three evenly spaced intermediate layers into the middle buffer, since they are required by DAV2 model head.

We test our example implementation on a stream of 30000 images of 1920×1080 resolution and it maintains² a 30 Hz throughput with median latencies of 20ms for depth estimation, 18ms for semantic segmentation, and 69ms for object detection on a Nvidia Orin AGX. It is important to note that the object detection head is not compiled with TensorRT, hence larger latency. Throughout the test, GPU memory usage was constant at around 1.5GB. Using VPEngine, the combined number of parameters of our example implementation is 27M, which 62% less than if we used three independent models, one for each task (71M parameters total).

²Due to freshness-oriented design and stochastic nature of the engine minimal losses are unavoidable, especially when using models not compiled with TensorRT. However, even then they account for $\approx 0.02\%$ of all frames.

V. BENCHMARK

To validate the design choices and quantify performance, we conducted two experiments. First, we compared our single multi-head model design to a series of independent, task-specific models. Second, we quantified the performance difference between a multiprocessing architecture and a single-process design. Both experiments used processing speed/latency and peak GPU memory usage as primary metrics. To measure processing speed, we measure the time from when an input image was inserted into the CPU array to the moment all heads/ models returned a CPU tensor³. All experiments were conducted under identical conditions using an NVIDIA Jetson AGX Orin 64GB.

A. Unified Backbone vs. Independent Models

This experiment highlights the benefit of using a unified backbone for multiple tasks instead of separate, full models. We compared VPEngine against two setups:

- **DepthAnythingV2:** A sequence of complete, independent DepthAnythingV2 models (each with its own backbone), compiled with TensorRT.
- **Sequential Heads:** A single, shared DINOv2 backbone followed by a sequence of TensorRT-optimized heads running one after another in the same process.

As shown in Figure 3, using a shared foundation model provides a significant speedup of up to **2.3×** compared to running multiple independent models. The performance of VPEngine is nearly identical to the “Sequential Heads” setup. This is because the TensorRT-optimized kernels used in this test already achieve nearly 100% GPU utilization, leaving little room for parallel processing to provide additional benefits. VPEngine introduces a negligible overhead of only about 2ms, which is due to the communication between processes.

Regarding memory, the full DepthAnythingV2 model uses an additional 96MB of GPU memory for each new task, the Sequential Heads baseline uses 48MB, and VPEngine uses 214MB per task.

B. Parallel vs. Sequential Processing

This experiment quantifies the advantage of VPEngine’s ability to run multiple model heads in parallel. We compared our framework against a “Sequential Object Detection Heads” baseline, which uses a single backbone but runs multiple PyTorch-based object detection heads one after another. Unlike the TensorRT kernels in the previous experiment, these PyTorch heads do not fully utilize the GPU.

The results in Figure 4 clearly show that VPEngine’s parallel, multi-process architecture is up to **3.3×** faster. This speedup comes from running several less-optimized kernels at the same time, which leads to much better overall GPU utilization. The importance of this design is further highlighted by the fact that disabling CUDA MPS resulted in a 43.4% loss in speed.

³This is only an approximation, or rather a lowerbound on the engine’s throughput, because it limits engine’s parallelization capabilities across inputs.

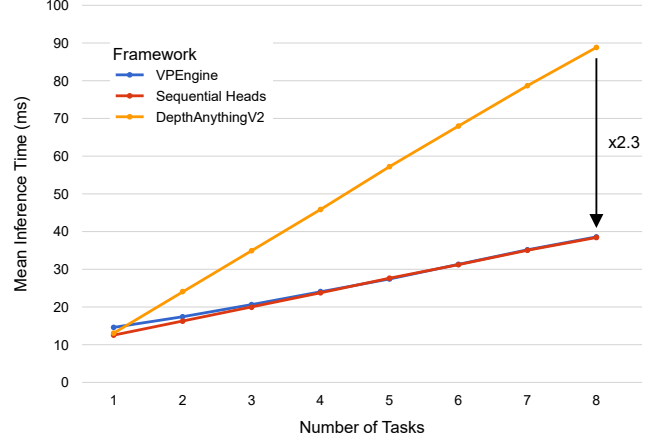


Fig. 3. Mean inference time across different number of tasks over 426 test images (of size 1920×1080). The arrow indicates $2.3\times$ speed gain of a single backbone with a sequence of 8 TensorRT heads over 8 sequential DepthAnythingV2 models.

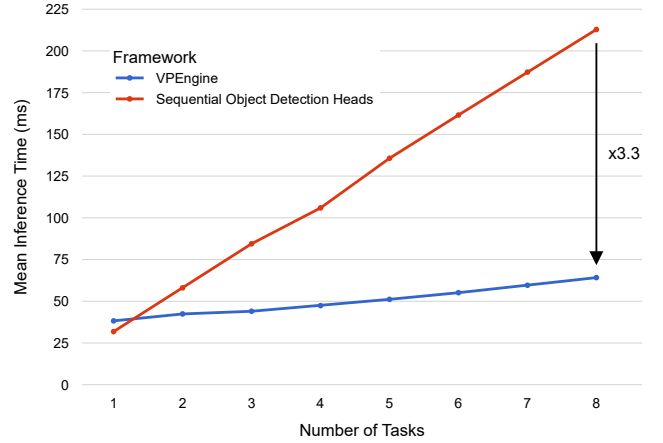


Fig. 4. Mean inference time across different number of tasks over 426 test images (of size 1920×1080). The arrow indicates $3.3\times$ speed gain of VPEngine over a single TensorRT foundation model with 8 PyTorch object detection heads.

This performance gain comes with a trade-off in memory usage. VPEngine requires about 295MB of GPU memory for each additional object detection head, whereas the sequential baseline only needs about 5MB. This higher memory footprint is because our multi-process design requires a separate instance of PyTorch (which uses ≈ 110 MB) to be loaded for each process. Each instance then manages its own separate block of GPU memory, unlike a single-process application that can reuse the same memory block for sequential tasks.

VI. LIMITATIONS

The most significant limitation of our framework is the use of multiple processes, leading to higher CPU usage. This also results in slightly higher GPU memory usage⁴.

⁴In our benchmarks we are using relatively small models, hence extra 295MB seems large in comparison, but please note that many models such as DINOv2-Giant use order of magnitude more memory (4.4GB in this case).

Secondly, different modules are running independently, which leads to overall stochastic nature of VPEngine. Implementing coordination between different processes, could potentially result in even higher GPU usage. We leave this to future work.

VII. CONCLUSION

We present VPEngine, a framework that offers substantial speedups through efficient GPU usage while still providing significant architecture flexibility. For example, it speeds up execution of Pytorch models by 3.3 times compared to running task-specific model heads sequentially.

This comes with a trade-off of higher CPU usage and slightly higher GPU memory usage. As a consequence, this might not be a low resource framework, however, for example, if a robotic stack is not using its full CPU capacity, the speedup offered may outweigh the tradeoffs.

REFERENCES

- [1] S. Thrun, W. Burgard, and D. Fox, *Probabilistic robotics*. MIT press, 2005.
- [2] B. Siciliano and O. Khatib, *Springer handbook of robotics*. Springer, 2016.
- [3] P. Corke, *Robotics, vision and control: fundamental algorithms In MATLAB*. Springer, 2017, vol. 118.
- [4] M. Oquab, T. Darcet, T. Moutakanni, H. V. Vo, M. Szafraniec, V. Khalidov, P. Fernandez, D. HAZIZA, F. Massa, A. El-Nouby, M. Assran, N. Ballas, W. Galuba, R. Howes, P.-Y. Huang, S.-W. Li, I. Misra, M. Rabbat, V. Sharma, G. Synnaeve, H. Xu, H. Jegou, J. Mairal, P. Labatut, A. Joulin, and P. Bojanowski, “DINOv2: Learning robust visual features without supervision,” *Transactions on Machine Learning Research*, 2024, featured Certification. [Online]. Available: <https://openreview.net/forum?id=a68SUt6zFt>
- [5] A. Kirillov, E. Mintun, N. Ravi, H. Mao, C. Rolland, L. Gustafson, T. Xiao, S. Whitehead, A. C. Berg, W.-Y. Lo *et al.*, “Segment anything,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2023, pp. 4015–4026.
- [6] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark *et al.*, “Learning transferable visual representations from natural language supervision,” in *International conference on machine learning*. PMLR, 2021, pp. 8748–8763.
- [7] NVIDIA Corporation, “NVIDIA Isaac ROS,” <https://developer.nvidia.com/isaac/ros>, 2025, accessed: 2025-05-16.
- [8] —, *Multi-Process Service (MPS)*, Online; accessed 20 July 2025, NVIDIA, Santa Clara, CA, 2025, version 575. [Online]. Available: <https://docs.nvidia.com/deploy/mps/index.html>
- [9] L. Yang, B. Kang, Z. Huang, Z. Zhao, X. Xu, J. Feng, and H. Zhao, “Depth anything v2,” in *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. [Online]. Available: <https://openreview.net/forum?id=cFTi3gIJ1X>
- [10] spacewalk01, “Depth-anything-tensorrt: Tensorrt implementation of depth-anything v1 and v2,” <https://github.com/spacewalk01/depth-anything-tensorrt>, 01 2024, gitHub repository.

APPENDIX A MULTIHEADED DESIGN - GPU RESULTS

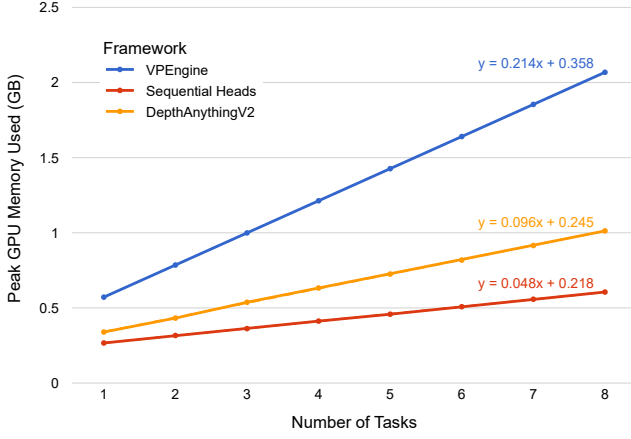


Fig. 5. Peak GPU memory used across different number of tasks over 426 test images (of size 1920×1080). Equations represent linear trend lines. Note that all the trend lines are plotted using dashed lines of respective colors, but they fully overlap with the datapoints.

APPENDIX B PARALLEL DESIGN - GPU RESULTS

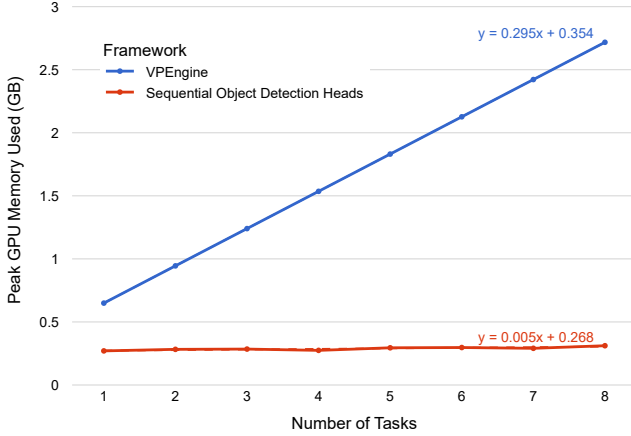


Fig. 6. Peak GPU memory used across different number of tasks over 426 test images (of size 1920×1080). Equations represent OLS trend lines for every. Note that all the trend lines are plotted using dashed lines of respective colors, but they fully overlap with the datapoints.

APPENDIX C CUDA MPS ABLATION RESULTS

We ablate the gain from using CUDA MPS by using the VPEngine setup from Section ?? and measuring the speed with CUDA MPS first disabled and then enables. Results are in Table.I

TABLE I
CUDA MPS ABLATION - IMAGE PROCESSING THROUGHPUT

Number of tasks	CUDA MPS Disabled (Hz)	CUDA MPS Enabled (Hz)	Relative Speedup
4	15.24	21.23	39.3%
8	8.77	15.52	77.0%

APPENDIX D RATE LIMITING EXPERIMENT

We conduct an experiment to test rate limiting feature of our engine. Our setup involves a stream of 300 images published at 30Hz over 10 seconds, where all 4 components of our example implementation (Section IV) are set to the same inference *rate*. Ultimately, we measured how many outputs were delivered by each head. Table II contains the results, which show that our rate limiting feature works with ± 1 margin of error.

TABLE II
VPENGINE RATE LIMITING RESULTS. NUMBERS IN BRACKETS CONTAIN IDEAL AMOUNT OF PROCESSED FRAMES, FOLLOWED BY THE ERROR.

Inference Rate (Hz)	Semantic Segmentation	Object Detection	Depth Estimation
5	(50) +1	(50) +1	(50) +1
10	(100) +1	(100) +1	(100) +1
15	(150) +1	(150) +1	(150) +1
20	(200) +1	(200) +1	(200) +1
25	(250) +1	(250) +1	(250) +1
30	(300) =0	(300) -1	(300) =0